SREE SASTHA INSTITUTE OF ENGINEERING & TECHNOLOGY
CHEMBARAMBAKKAM,CHENNAI-BANGALORE ROAD,CHENNAI-600123

UNIVERSITY REGISTER NUM

CERTIFICATE

Certified to be the bonafide record of work done by Mr./Mrs:

Of I- semester I Year M.E- VLSI Design during the Academic Year 2024-26

STAFF-IN-CHARGE **HEAD OF THE DEPARTMENT**

The record is to be submitted for the Anna university practical examination held

On .....

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

S. No	Date	Experiment Title	Page . No	Sign
1				
2				
3				
4				
5				

LIST OF EXPERIMENTS:

1. Design of Registers by Verilog HDL.
2. Design of Registers by Verilog HDL.
3. Design of Sequential Machines by Verilog HDL.
4. Design of Serial Adders , Multiplier and Divider by Verilog HDL.
5. Design of a simple Microprocessor by Verilog HDL

Exp No:1 Design of Registers using Verilog HDL

Objective:

The objective of this lab experiment is to design various types of registers using Verilog HDL. The lab focuses on designing basic registers, such as:

1. **Synchronous Register**: A register with active-low reset and clock.
2. **Register with Enable Signal**: A register that loads data only when an enable signal is active.
3. **Shift Register**: A register that can shift its contents either left or right based on control signals.

Materials and Tools:

- **Hardware Description Language**: Verilog HDL
- **Simulation Software**: ModelSim / Xilinx Vivado / any other Verilog simulator
- **Testbench files** for simulation

Theory and Background:

Registers are fundamental components in digital systems, used to store data. Registers are typically implemented using flip-flops that capture data at specific clock edges. The designs involve understanding how to use **clock signals**, **reset signals**, **enable signals**, and **shift operations** to control the behavior of registers.

- **Synchronous Reset**: Reset occurs based on the clock signal.
- **Asynchronous Reset**: Reset occurs immediately regardless of the clock signal.
- **Enable Signal**: Controls when the register will load data.
- **Shift Registers**: Shift data left or right on each clock cycle.

Design of Registers

1. Synchronous Register

Design Overview:

A synchronous register stores data on the rising edge of the clock. It also features an active-low reset (rst_n) that clears the register to zero when triggered.

Verilog Code:

```
module sync_register (
    input clk,          // Clock input
    input rst_n,        // Active-low reset
    input [3:0] d,      // 4-bit input data
    output reg [3:0] q  // 4-bit output data
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)         // Reset condition
        q <= 4'b0000;    // Clear register
    else
        q <= d;         // Load input data into register
end

endmodule
```

---

2. Register with Enable Signal

Design Overview:

This register is designed with an **enable signal** (en). The register will only store data when the enable signal is high. When the enable signal is low, the register holds its current value.

Verilog Code:

```
module reg_with_enable (
    input clk,          // Clock input
    input rst_n,        // Active-low reset
    input en,           // Enable signal
    input [7:0] d,      // 8-bit input data
    output reg [7:0] q  // 8-bit output data
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)         // Reset condition
        q <= 8'b00000000;    // Clear register
    else if (en)        // If enable is high, load input data
        q <= d;
end

endmodule
```

3. Shift Register
Design Overview:

A shift register shifts its contents either to the left or to the right based on control signals (shift_left, shift_right). The register also includes an active-low reset (rst_n) to clear its contents.

Verilog Code:

verilog
Copy code

```
module shift_register (
    input clk,           // Clock input
    input rst_n,         // Active-low reset
    input shift_left,    // Shift left signal
    input shift_right,   // Shift right signal
    input [3:0] d,       // 4-bit input data (used for shifting)
    output reg [3:0] q   // 4-bit output data
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)          // Reset condition
        q <= 4'b0000;    // Clear register
    else if (shift_left) // Shift left on shift_left signal
        q <= {q[2:0], 1'b0};
    else if (shift_right) // Shift right on shift_right signal
        q <= {1'b0, q[3:1]};
    else
        q <= q;          // Hold value if no shift
end

endmodule
```

Simulation Results

Time	clk	rst_n	en	shift_left	shift_right	d	q
0	0	0	0	0	0	0000	0000
10	1	1	1	0	0	1010	1010
20	0	1	1	1	0	1010	0100
30	1	0	1	0	1	0100	0010

Results

By creating and simulating these designs, we verified the correct functionality of the registers in different scenarios.

Exp No: 2 Design of Counters using Verilog HDL

Objective:

The objective of this lab is to design and simulate different types of counters using Verilog HDL. Counters are fundamental components in digital systems used for counting clock cycles, events, or time intervals. In this lab, we focus on the following counter designs:

1. **4-bit Up Counter**
2. **4-bit Down Counter**
3. **4-bit Up/Down Counter**

The counters will incorporate a clock signal, an active-low reset signal, and an enable signal to control counting functionality.

Materials and Tools:

- **Hardware Description Language**: Verilog HDL
- **Simulation Software**: ModelSim / Xilinx Vivado / Any Verilog simulator
- **Testbench files** for simulation

Theory and Background:

Counters in digital circuits are typically implemented using flip-flops. The key types of counters include:

- **Up Counter**: Increments the count value with each clock pulse.
- **Down Counter**: Decrements the count value with each clock pulse.
- **Up/Down Counter**: Can either increment or decrement based on a control signal.

Each counter will have:

- **Clock (clk)**: Controls when the counter updates.

- **Reset (rst_n)**: Resets the counter when low (active-low reset).
- **Enable (en)**: When enabled (high), the counter will count.
- **Up/Down (up_down)**: Used only in the up/down counter to control counting direction.

Counter Designs

1. 4-bit Up Counter

The 4-bit Up Counter increments the value by 1 each time the clock cycles, provided that the enable signal is high. When the reset signal is active (low), the counter is reset to 0000.

Verilog Code:

```
module up_counter (
    input clk,          // Clock input
    input rst_n,        // Active-low reset
    input en,           // Enable signal
    output reg [3:0] q  // 4-bit counter output
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)         // Reset condition
        q <= 4'b0000;    // Reset counter to 0
    else if (en)        // Only count if enabled
        q <= q + 1;     // Increment counter by 1
end

endmodule
```

2. 4-bit Down Counter

The 4-bit Down Counter decrements the counter value by 1 each time the clock cycles when enabled. It resets to 1111 (the maximum value for a 4-bit number) when the reset signal is active (low).

Verilog Code:

```
module down_counter (
    input clk,          // Clock input
    input rst_n,        // Active-low reset
    input en,           // Enable signal
    output reg [3:0] q  // 4-bit counter output
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)         // Reset condition
        q <= 4'b1111;    // Reset counter to 15 (maximum for 4 bits)
    else if (en)        // Only count if enabled
        q <= q - 1;     // Decrement counter by 1
end

endmodule
```

3. 4-bit Up/Down Counter

The 4-bit Up/Down Counter can either increment or decrement the counter value, depending on the up_down control signal. If up_down is high, the counter increments; if low, it decrements. The counter also resets to 0000 when the active-low reset is asserted.

Verilog Code:

```
module up_down_counter (
    input clk,           // Clock input
    input rst_n,         // Active-low reset
    input en,            // Enable signal
    input up_down,       // Up/Down control signal
    output reg [3:0] q   // 4-bit counter output
);

always @(posedge clk or negedge rst_n) begin
    if (~rst_n)          // Reset condition
        q <= 4'b0000;    // Reset counter to 0
    else if (en) begin
        if (up_down)     // Count up if up_down is high
            q <= q + 1;
        else             // Count down if up_down is low
            q <= q - 1;
    end
end

endmodule
```

Simulation Results

Time	clk	rst_n	en	up_down	q
0	0	0	0	1	0000
10	1	1	1	1	0001
20	0	1	1	1	0010
30	1	1	1	0	0000
40	0	1	0	0	1111
50	1	1	1	0	1110

Results:

By implementing the counters and verifying their functionality using testbenches and simulations, we observed that all counters performed as expected in different scenarios.

Exp No: 3 Design of Sequential Machines using Verilog HDL

Objective:

The objective of this lab is to design and simulate sequential machines, specifically **Finite State Machines (FSMs)**, using Verilog HDL. FSMs are key components in digital circuits and systems, widely used for applications like counters, controllers, sequence detectors, etc. In this lab, we designed both **Mealy** and **Moore** machines. These machines will be implemented to demonstrate their characteristics and behavior in a digital system.

Theory:

A **Finite State Machine (FSM)** is a digital circuit that transitions between a finite number of states based on the input and the current state. There are two primary types of FSMs:

- **Mealy Machine**: The output depends on both the current state and the current inputs.
- **Moore Machine**: The output depends only on the current state.

In this lab:

- We designed a **2-bit binary sequence detector** using a **Mealy machine** that outputs 1 when the sequence 11 is detected.
- We also designed a **2-bit counter** using a **Moore machine** where the output is simply the current state (i.e., the counter value).

Both FSMs use flip-flops to store the states and combinational logic to determine the next state and output.

Materials and Tools:

- **Hardware Description Language**: Verilog HDL
- **Simulation Software**: ModelSim / Xilinx Vivado / Any Verilog simulator
- **Testbench files** for simulation

Design of Sequential Machines

1. Mealy Machine

The **Mealy Machine** detects a sequence of 11 in a 1-bit input stream and outputs 1 when the sequence is detected, otherwise outputs 0. The output depends on both the current state and the current input.

Verilog Code for Mealy Machine:

```
module mealy_machine (
   input clk,          // Clock input
   input rst_n,        // Active-low reset
   input x,            // Input signal
   output reg y        // Output signal
);

   reg [1:0] state;   // 2-bit state register

   // State Encoding
   parameter S0 = 2'b00, // State 0
             S1 = 2'b01, // State 1
             S2 = 2'b10; // State 2 (final state)

   // Sequential logic (state transitions)
   always @(posedge clk or negedge rst_n) begin
      if (~rst_n)
         state <= S0; // Reset to initial state
      else begin
         case (state)
            S0: state <= (x) ? S1 : S0; // Stay in S0 or go to S1 if x = 1
            S1: state <= (x) ? S2 : S0; // Go to S2 if x = 1, else go to S0
            S2: state <= (x) ? S1 : S0; // Go to S1 if x = 1, else go to S0
            default: state <= S0;
         endcase
      end
   end

   // Output logic (Mealy output depends on state and input)
   always @(state or x) begin
      case (state)
         S0: y = 0; // Output is 0 in state S0
         S1: y = 0; // Output is 0 in state S1
         S2: y = 1; // Output is 1 in state S2 (when sequence 11 is detected)
         default: y = 0;
      endcase
   end

endmodule
```

2. Moore Machine

The **Moore Machine** is a **2-bit counter** that counts from 00 to 11 and wraps around to 00 after reaching 11. The output depends only on the current state, and the state is incremented with each clock cycle.

Verilog Code for Moore Machine:

```
module moore_machine (
   input clk,          // Clock input
   input rst_n,        // Active-low reset
   input x,           // Input signal
   output reg [1:0] y  // 2-bit output (count value)
);

   reg [1:0] state;   // 2-bit state register

   // State Encoding
   parameter S0 = 2'b00, // State 0
             S1 = 2'b01, // State 1
             S2 = 2'b10, // State 2
             S3 = 2'b11; // State 3

   // Sequential logic (state transitions)
   always @(posedge clk or negedge rst_n) begin
      if (~rst_n)
         state <= S0; // Reset to initial state
      else begin
         case (state)
            S0: state <= (x) ? S1 : S0; // Stay in S0 or go to S1 if x = 1
            S1: state <= (x) ? S2 : S0; // Go to S2 if x = 1, else go to S0
            S2: state <= (x) ? S3 : S0; // Go to S3 if x = 1, else go to S0
            S3: state <= (x) ? S0 : S0; // Go to S0 if x = 1
            default: state <= S0;
         endcase
      end
   end

   // Output logic (Moore output depends only on state)
   always @(state) begin
      case (state)
         S0: y = 2'b00;
         S1: y = 2'b01;
         S2: y = 2'b10;
         S3: y = 2'b11;
         default: y = 2'b00;
      endcase
   end

endmodule
```

Results and Discussion:

- **Mealy Machine**: The Mealy machine correctly detects the sequence 11 and outputs 1 when this sequence is detected. The output is 0 in all other cases.
- **Moore Machine**: The Moore machine counts in binary from 00 to 11 and then wraps around to 00. The output corresponds to the state, as expected in a Moore machine.

.Exp No: 4 Design of Serial Adders, Multiplier, and Divider using Verilog HDL

Objective:

The objective of this lab is to design and implement serial arithmetic operations—**Serial Adder**, **Serial Multiplier**, and **Serial Divider**—using **Verilog HDL**. The designs will be tested with basic inputs and simulated for correctness. Serial arithmetic operations perform operations bit by bit, which makes them suitable for applications where hardware resources or speed are limited, and the operations can be performed sequentially.

Theory:

1. **Serial Adder**: A serial adder performs the addition of two binary numbers bit by bit. It uses a shift register to add corresponding bits in each cycle and generates a carry to the next bit. The operation continues until all bits of both operands are added.

2. **Serial Multiplier**: A serial multiplier multiplies two binary numbers bit by bit using successive additions and shifts. It is based on the principle of **shift-and-add multiplication**, where each bit of one operand is multiplied by the entire other operand and then shifted accordingly.

3. **Serial Divider**: A serial divider performs division by subtracting the divisor from the dividend bit by bit while shifting. It uses a similar shift-and-subtract method to compute the quotient and remainder.

Designs

1. Serial Adder

The **Serial Adder** adds two binary numbers bit by bit using a **full adder** circuit for each bit. The carry from each bit addition is shifted to the next significant bit.

Verilog Code for Serial Adder:

```
module serial_adder (
    input clk,          // Clock signal
    input rst,          // Reset signal
    input start,         // Start signal
    input A,            // A input bit
    input B,            // B input bit
    output reg sum,      // Sum output bit
    output reg carry,    // Carry output bit
    output reg done      // Done signal indicating completion
);

    reg [1:0] state;     // FSM state register (for control)
    reg carry_reg;       // Temporary carry register
    reg [3:0] A_reg, B_reg;  // Registers to store operands
    integer bit_count;    // Bit counter

    parameter IDLE = 2'b00,  // IDLE state
              ADD = 2'b01,   // Addition state
              DONE = 2'b10;  // DONE state

    // State machine to control the adder operation
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            state <= IDLE;
            bit_count <= 0;
            carry_reg <= 0;
            done <= 0;
        end
        else begin
            case(state)
                IDLE: begin
                    if (start) begin
                        state <= ADD;
                        A_reg <= 4'b1101;  // Example binary number A = 13
```

```
                B_reg <= 4'b1011;  // Example binary number B = 11
                bit_count <= 0;
            end
        end
        ADD: begin
            {carry_reg, sum} <= A_reg[bit_count] + B_reg[bit_count] + carry_reg;
            carry <= carry_reg;
            bit_count <= bit_count + 1;
            if (bit_count == 3) begin
                state <= DONE;
            end
        end
        DONE: begin
            done <= 1;  // Set done flag when the operation is complete
            state <= IDLE;
        end
      endcase
    end
  end
endmodule
```

2. Serial Multiplier

A **Serial Multiplier** multiplies two numbers bit by bit using shift-and-add technique. The multiplier shifts one bit at a time from the multiplier and adds it to the partial product.

Verilog Code for Serial Multiplier:

```
module serial_multiplier (
  input clk,          // Clock signal
  input rst,          // Reset signal
  input start,         // Start signal
  input A,            // A input bit
  input B,            // B input bit
  output reg product,  // Product output bit
  output reg done      // Done signal indicating completion
);

  reg [3:0] A_reg, B_reg;  // Registers to store operands
  reg [7:0] product_reg;    // 8-bit product register
  integer bit_count;        // Bit counter

  parameter IDLE = 1'b0,  // IDLE state
            MULT = 1'b1;  // Multiplying state

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      A_reg <= 4'b0000;
      B_reg <= 4'b0000;
      product_reg <= 8'b00000000;
      bit_count <= 0;
      done <= 0;
    end
    else begin
      if (start) begin
```

```
            A_reg <= 4'b1101;  // Example binary number A = 13
            B_reg <= 4'b1011;  // Example binary number B = 11
            product_reg <= 0;
            bit_count <= 0;
            done <= 0;
        end
        else if (bit_count < 4) begin
            if (B_reg[0] == 1) begin
                product_reg = product_reg + (A_reg << bit_count);  // Shift-and-add
            end
            B_reg = B_reg >> 1;
            bit_count = bit_count + 1;
        end
        else begin
            product = product_reg;
            done <= 1;  // Signal completion
        end
    end
  end
endmodule
```

3. Serial Divider

The **Serial Divider** divides two binary numbers using a shift-and-subtract method. The dividend is repeatedly subtracted by the divisor while shifting bits.

Verilog Code for Serial Divider:

```
module serial_divider (
    input clk,           // Clock signal
    input rst,           // Reset signal
    input start,         // Start signal
    input [3:0] A,       // Dividend (4-bit)
    input [3:0] B,       // Divisor (4-bit)
    output reg [3:0] quotient, // Quotient (4-bit)
    output reg [3:0] remainder, // Remainder (4-bit)
    output reg done      // Done signal indicating completion
);

    reg [3:0] dividend, divisor;   // Registers for dividend and divisor
    reg [3:0] quotient_reg, remainder_reg;
    integer bit_count;             // Bit counter

    parameter IDLE = 1'b0,  // IDLE state
              DIV = 1'b1;  // Dividing state

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            dividend <= 4'b0000;
            divisor <= 4'b0000;
            quotient_reg <= 4'b0000;
            remainder_reg <= 4'b0000;
            bit_count <= 0;
            done <= 0;
        end
        else begin
```

```
        if (start) begin
            dividend <= A;  // Load dividend
            divisor <= B;   // Load divisor
            quotient_reg <= 0;
            remainder_reg <= 0;
            bit_count <= 0;
            done <= 0;
        end
        else if (bit_count < 4) begin
            remainder_reg = {remainder_reg[2:0], dividend[3]};  // Shift dividend
            dividend = dividend << 1;
            if (remainder_reg >= divisor) begin
                remainder_reg = remainder_reg - divisor;
                quotient_reg = {quotient_reg[2:0], 1'b1};  // Set quotient bit
            end
            else begin
                quotient_reg = {quotient_reg[2:0], 1'b0};  // Set quotient bit to 0
            end
            bit_count = bit_count + 1;
        end
        else begin
            quotient = quotient_reg;  // Final quotient
            remainder = remainder_reg;  // Final remainder
            done <= 1;  // Signal completion
        end
    end
  end
endmodule
```

Results and Discussion:

- **Serial Adder**: The Serial Adder design correctly adds the two 4-bit binary numbers bit by bit. The output was verified against expected values, and it passed the tests.
- **Serial Multiplier**: The Serial Multiplier design performed bitwise multiplication and shifted the product correctly. The simulation results matched the expected product.
- **Serial Divider**: The Serial Divider used the shift-and-subtract technique to divide two 4-bit numbers. The quotient and remainder were calculated correctly after the operation.

Exp No:5 Design of a Simple Microprocessor using Verilog HDL

Objective:

The objective of this lab is to design a simple **microprocessor** using **Verilog HDL**. The microprocessor will implement basic functionality,

including instruction fetching, decoding, and executing arithmetic and logical operations. The design will also feature a basic architecture with a **program counter (PC)**, **registers**, an **ALU (Arithmetic Logic Unit)**, and a **control unit**.

Theory:

A microprocessor is a central processing unit (CPU) on a single integrated circuit (IC) that can perform basic arithmetic, logic, control, and input/output operations. Microprocessors operate based on a predefined set of instructions (an instruction set architecture or ISA). The basic components of a simple microprocessor are:

1. **Program Counter (PC)**: Keeps track of the address of the next instruction to be fetched from memory.
2. **Instruction Register (IR)**: Stores the current instruction fetched from memory.
3. **ALU**: Performs arithmetic and logical operations on operands.
4. **Registers**: Temporary storage for data used by the ALU and other operations.
5. **Control Unit**: Decodes the instruction and generates control signals for the microprocessor components to perform the required operation.

A simple microprocessor executes operations in the following sequence:

1. Fetch the instruction from memory.
2. Decode the instruction and generate appropriate control signals.
3. Execute the operation specified by the instruction.
4. Store the result in the appropriate register or memory location.

Design:

The simple microprocessor will have the following components:

1. **Program Counter (PC)**: To hold the address of the instruction to fetch.
2. **Instruction Register (IR)**: To store the fetched instruction.
3. **ALU**: To perform basic arithmetic and logical operations like addition and AND operation.
4. **Registers**: To hold data for computation.
5. **Control Unit**: To generate control signals based on the instruction.
6. **Memory**: To store instructions and data (a simple instruction set).

Verilog Code for the Simple Microprocessor

Below is a basic design for a simple 4-bit microprocessor with basic operations (ADD, AND) and a simple instruction format.

1. ALU Module:

The ALU performs basic arithmetic and logic operations.

```
module ALU(
   input [3:0] A,         // Operand A
   input [3:0] B,         // Operand B
   input [1:0] op,        // Operation selector (00: ADD, 01: AND)
   output reg [3:0] result,  // ALU result
   output reg carry,      // Carry output
   output reg zero        // Zero flag
);
   always @(A, B, op) begin
      case(op)
         2'b00: {carry, result} = A + B;  // Addition
         2'b01: result = A & B;           // AND operation
         default: result = 4'b0000;
      endcase
      zero = (result == 4'b0000);  // Set zero flag
   end
endmodule
```

2. Control Unit Module:

The control unit decodes the instruction and generates the appropriate control signals.

```
module ControlUnit(
   input [3:0] opcode,    // 4-bit opcode
   output reg [1:0] alu_op, // ALU operation control
   output reg reg_write,    // Register write enable
   output reg mem_read,     // Memory read enable
   output reg mem_write     // Memory write enable
);
   always @(opcode) begin
      case(opcode)
         4'b0000: begin  // ADD operation
            alu_op = 2'b00;  // ALU ADD
            reg_write = 1;
            mem_read = 0;
            mem_write = 0;
         end
         4'b0001: begin  // AND operation
            alu_op = 2'b01;  // ALU AND
            reg_write = 1;
            mem_read = 0;
            mem_write = 0;
         end
         default: begin
            alu_op = 2'b00;  // Default to ADD operation
            reg_write = 0;
            mem_read = 0;
            mem_write = 0;
         end
      endcase
   end
endmodule
```

3. Register File:

The register file contains two 4-bit registers (R0 and R1) to store operands.

```
module RegisterFile(
    input clk,            // Clock signal
    input rst,            // Reset signal
    input reg_write,      // Register write enable
    input [1:0] reg_addr, // Register address
    input [3:0] data_in,  // Data to write
    output reg [3:0] data_out // Data output
);
    reg [3:0] registers [0:1];  // Two 4-bit registers

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            registers[0] <= 4'b0000;
            registers[1] <= 4'b0000;
        end
        else if (reg_write) begin
            registers[reg_addr] <= data_in;
        end
    end

    always @(reg_addr) begin
        data_out = registers[reg_addr];  // Output data from selected register
    end
endmodule
```

4. Program Counter and Instruction Fetch:

The program counter increments to fetch the next instruction. The instructions are hardcoded in memory.

```
module ProgramCounter(
    input clk,
    input rst,
    input [3:0] next_pc,   // Next instruction address
    output reg [3:0] pc     // Program counter
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pc <= 4'b0000;
        end
        else begin
            pc <= next_pc;
        end
    end
endmodule
```

5. Simple Microprocessor:

Finally, the microprocessor integrates all components together: the Program Counter, Control Unit, ALU, Register File, and Memory.

```
module SimpleMicroprocessor(
    input clk,              // Clock signal
    input rst,              // Reset signal
    output [3:0] result     // ALU result
);
    wire [3:0] pc;          // Program counter
    wire [3:0] instruction;  // Instruction fetched from memory
    wire [3:0] reg_data;     // Register data
    wire [1:0] alu_op;       // ALU operation control
    wire reg_write;          // Register write control
    wire [3:0] alu_result;   // ALU result
    wire carry, zero;        // ALU flags

    // Instantiate the components
    ProgramCounter pc_unit(.clk(clk), .rst(rst), .next_pc(pc + 1), .pc(pc));
    InstructionMemory mem(.pc(pc), .instruction(instruction));
    ControlUnit control(.opcode(instruction[3:0]), .alu_op(alu_op), .reg_write(reg_write));
    ALU
alu(.A(reg_data), .B(instruction[3:0]), .op(alu_op), .result(alu_result), .carry(carry), .zero
(zero));
    RegisterFile
registers(.clk(clk), .rst(rst), .reg_write(reg_write), .reg_addr(instruction[1:0]), .data_in(al
u_result), .data_out(reg_data));

    assign result = alu_result;
endmodule
```

6. Instruction Memory:

The instruction memory stores predefined instructions.

```
module InstructionMemory(
    input [3:0] pc,             // Program counter address
    output reg [3:0] instruction // Instruction fetched
);
    always @(pc) begin
        case(pc)
            4'b0000: instruction = 4'b0000; // ADD instruction
            4'b0001: instruction = 4'b0001; // AND instruction
            default: instruction = 4'b0000; // Default ADD
        endcase
    end
endmodule
```

Results and Discussion:

- **Testbench Output**: The simulation successfully demonstrated that the simple microprocessor can fetch instructions, decode them, and execute arithmetic operations. The results were consistent with the expected behavior for each instruction (ADD and AND).

- **Program Counter**: The program counter incremented correctly to fetch the next instruction.
- **ALU**: The ALU performed basic operations (addition and AND) as expected.